100 Core Concepts for `PendulumSystem.jl` and ModelingToolkit.jl

Self-learning Notes

How to Use These 100 Concepts

These 100 items are organized as a structured learning map for building and understanding a ModelingToolkit.jl pendulum package (`PendulumSystem.jl`) from scratch:

- Each concept has a short explanation.
- Many concepts include a small Julia snippet.
- You can treat them as checkpoints or flashcards.

They are grouped into themes:

A. Physics of the Pendulum

B. MTK Fundamentals

C. Minimal Pendulum `ODESystem`

D. From Symbolic System to Simulation

E. `@mtkmodel` and Reusable Components

F. Julia Package Structure

G. High-level Simulation API

H. Energy Analysis

I. Plotting Helpers

J. Damping Extension

K. Driven Pendulum Extension

L. Poincaré Map Concepts

M. Learning and Reasoning Patterns

1 A. Physics of the Pendulum (Concept → Code)

A1. Pendulum as a physical system

A mass m on a massless rod of length L under gravity g, moving in a vertical plane. The entire motion is captured by the angle $\theta(t)$.

No code yet, just the mental model: a single degree of freedom system.

A2. Angle θ as generalized coordinate

Instead of $x(t)$ and $y(t)$, we choose $\theta(t)$:

$$x = L\sin\theta, \quad y = -L\cos\theta,$$

so we only need θ.

```
@variables (t)
```

A3. Restoring torque proportional to $\sin(\theta)$

Component of gravity along the arc gives:

$$\tau = -mgL\sin\theta.$$

```
= -m * g * L * sin()
```

A4. Moment of inertia of point mass on rod

For a point mass at distance L:

$$I = mL^2.$$

```
I = m * L^2
```

A5. Core second-order pendulum ODE

From $I\ddot{\theta} = \tau$:

$$\ddot{\theta} = -\frac{g}{L}\sin\theta.$$

```
= -(g/L) * sin()
```

A6. State splitting: θ and $\omega = \dot{\theta}$

Introduce angular velocity ω:

$$\dot{\theta} = \omega, \quad \dot{\omega} = -\frac{g}{L}\sin\theta.$$

```
=
= -(g/L) * sin()
```

A7. Why move to first-order form

General solver form:

$$\dot{x} = f(x, t)$$

is required by most ODE solvers and MTK; hence we split $\ddot{\theta}$ into two first-order equations.

A8. Torque always pulls back toward $\theta = 0$

Sign intuition:

- If $\theta > 0$, then $\sin\theta > 0$ and $\dot{\omega} < 0$.
- If $\theta < 0$, then $\sin\theta < 0$ and $\dot{\omega} > 0$.

The motion is always pulled back toward the stable equilibrium.

A9. Small-angle approximation $\sin\theta \approx \theta$

For small θ (in radians):

$$\sin\theta \approx \theta, \qquad \dot{\omega} \approx -\frac{g}{L}\theta.$$

This gives a linear pendulum model useful for analytic checks and validation.

A10. Pendulum as energy-conserving system (ideal case)

Without damping and forcing:

$$T = \tfrac{1}{2}mL^2\omega^2, \quad V = mgL(1-\cos\theta), \quad E = T + V.$$

```
T = 0.5 * m * L^2 * ^2
V = m * g * L * (1 - cos())
E = T + V
```

In the ideal model E is constant in time.

2 B. MTK Fundamentals (Variables, Parameters, Equations)

B1. MTK focuses on "math, not code"

You describe equations; MTK handles transformation and code generation:

```
@variables ...
@parameters ...
eqs = [...]
```

B2. Three pillars of an MTK system

Every model consists of:

- Variables (states),
- Parameters (constants),
- Equations (constraints).

```
pendulum = ODESystem(eqs, t, [, ], [g, L])
```

B3. Independent variable t

```
@variables t
D = Differential(t)
```

MTK needs to know with respect to which variable derivatives are taken.

B4. Differential operator `D`

```
D()  # symbolic representation of d/dt
```

`D` creates symbolic derivatives.

B5. Time-varying state declaration

```
@variables (t) (t)
```

The `(t)` marks them as functions of time.

B6. Constant parameter declaration

```
@parameters g L
```

Parameters are constant during a simulation (unless you explicitly model them as states).

B7. Do not declare parameters as time functions

Incorrect:

```
@variables g(t)
```

This makes g a state rather than a parameter.

B8. Equation syntax uses ~, not =

```
D() ~
```

Eqn. is interpreted as $D(\theta) - \omega = 0$ (constraint), not assignment.

B9. Definition equation vs physical law

```
D() ~                     # kinematic definition
D() ~ -(g/L) * sin()      # dynamic law
```

Separating these mentally helps with understanding index and structure.

B10. Equations as constraints, not instructions

The order of equations does not define execution order; MTK treats them as a system of constraints and reorders as needed.

B11. `ODESystem` as a symbolic container

```
pendulum = ODESystem(eqs, t, [, ], [g, L])
```

It stores symbolic variables, parameters, and equations, but no numerical solver is involved yet.

3 C. Minimal Pendulum `ODESystem`

C1. Minimal MTK pendulum model

```
@variables t
D = Differential(t)
@variables (t) (t)
@parameters g L

eqs = [
    D() ~ ,
    D() ~ -(g/L) * sin(),
]

pendulum = ODESystem(eqs, t, [, ], [g, L])
```

A complete symbolic model with 2 equations and 2 states.

C2. Equation / variable count sanity check

```
length(eqs) == length([, ])
```

Quick check for a square system.

C3. Independent variable in `ODESystem`

```
pendulum = ODESystem(eqs, t, [, ], [g, L])
```

The `t` argument tells MTK what the independent variable is.

C4. Separation of model vs simulation

The `ODESystem` is purely symbolic; solving it happens later using `ODEProblem` and `solve`.

4 D. From Symbolic System to Simulation

D1. Structural simplification

```
sys = structural_simplify(pendulum)
```

MTK rewrites the system into a more explicit, solvable form (possibly with fewer equations and variables).

D2. Initial condition mapping using pairs

```
u0 = [
     => /6,
     => 0.0
]
```

Symbolically attach numerical initial conditions to states.

D3. Parameter mapping

```
p = [
    g => 9.81,
    L => 1.0
]
```

Bind physical parameter values for the simulation.

D4. Time span for integration

```
tspan = (0.0, 10.0)
```

Finite interval for the ODE solver.

D5. Creating an `ODEProblem`

```
prob = ODEProblem(sys, u0, tspan, p)
```

This connects symbolic model, initial conditions, time span, and parameters.

D6. Solving the problem

```
sol = solve(prob)
```

Produces a solution object that stores time points and state values.

D7. Solution object fields

```
sol.t      # time vector
sol[]     # (t) samples
sol[]     # (t) samples
```

Symbolic indexing by variables is supported.

D8. Plotting states vs time

```
using Plots
plot(sol, vars=[, ])
```

High-level plotting of solution variables.

D9. Explicit plotting with arrays

```
plot(sol.t, sol[], label="(t)")
```

Gives more control over labels and styles.

D10. Simulation pipeline mental model

$$\text{ODESystem} \rightarrow \text{structural\_simplify} \rightarrow \text{ODEProblem} \rightarrow \text{solve} \rightarrow \text{plot}.$$

This workflow generalizes to many future models.

5 E. `@mtkmodel` and Reusable Components

E1. Component-style modeling with `@mtkmodel`

```
@mtkmodel Pendulum begin
    @parameters ...
    @variables ...
    @equations ...
end
```

Encapsulates a model as a reusable component.

E2. Parameter block with defaults

```
@parameters begin
    g = 9.81
    L = 1.0
end
```

Defaults make the component ready to use without extra configuration.

E3. Variables block inside `@mtkmodel`

```
@variables begin
    (t)
    (t)
end
```

Declares all time-dependent variables for the component.

E4. Equations block inside `@mtkmodel`

```
@equations begin
    D() ~
    D() ~ -(g/L)*sin()
end
```

Collects all physics in one block.

E5. Implicit time & differential in `@mtkmodel`

Within `@mtkmodel`, you do not explicitly redeclare `t` and `D`; they are inserted for you.

E6. Constructing a component instance

```
pend = Pendulum()
```

Creates a symbolic component instance.

E7. Overriding parameter defaults at construction

```
pend = Pendulum(g = 9.81, L = 2.0)
```

This pattern is very convenient for parameter studies.

E8. Multi-level models with `@components` (later)

```
@components begin
    p1 = Pendulum()
    p2 = Pendulum()
end
```

Used for composite systems (e.g., double pendulum).

6 F. Julia Package Structure (`PendulumSystem.jl`)

F1. Standard `src` layout

```
PendulumSystem.jl/
  src/
    PendulumSystem.jl
    pendulum.jl
    simulate.jl
    energy.jl
    plots.jl
```

Organizes code into logical files.

F2. Main module file

```
module PendulumSystem
    using ModelingToolkit, DifferentialEquations, Plots

    include("pendulum.jl")
    include("simulate.jl")
    include("energy.jl")
```

```
    include("plots.jl")

    export Pendulum, simulate_pendulum,
           compute_energy, plot_energy, plot_phase
end
```

Defines the public API of the package.

F3. Keeping models in their own file

```
src/pendulum.jl  # contains @mtkmodel Pendulum
```

Improves modularity.

F4. Placing simulations in `simulate.jl`

```
# src/simulate.jl
function simulate_pendulum(...)
    ...
end
```

Groups user-facing simulation entry points.

F5. Export only what users need

```
export Pendulum, simulate_pendulum,
       compute_energy, plot_energy, plot_phase
```

Keeps the namespace tidy and focused.

7 G. High-Level Simulation API (`simulate_pendulum`)

G1. User-friendly function signature

```
function simulate_pendulum(; 0=/6, 0=0.0,
                             g=9.81, L=1.0,
                             tspan=(0.0, 10.0))
```

Keyword arguments with sensible defaults.

G2. Constructing the component inside the function

```
pend = Pendulum(g=g, L=L)
```

Hides internal details from users.

G3. Simplification inside the simulate function

```
sys = structural_simplify(pend)
```

Ensures users get a clean system.

G4. Embedding parameters in the component

Because parameters are stored in `Pendulum`, you can often omit parameter vectors for the problem:

```
prob = ODEProblem(sys, u0, tspan)
```

G5. Build & solve inside helper

```
prob = ODEProblem(sys, u0, tspan)
sol  = solve(prob)
return sol
```

Users only have to call `simulate_pendulum`; they do not need to work with `ODEProblem` directly.

G6. Internal initial condition construction

```
u0 = [ => 0,  => 0]
```

Maps physical initial conditions to states.

G7. Single entry point pattern

```
sol = simulate_pendulum()
```

Provides a one-line experiment.

8 H. Energy Analysis (`compute_energy`)

H1. Kinetic energy from ω

```
T = 0.5 * m * L^2 .* (_values .^ 2)
```

Computed elementwise from solution data.

H2. Potential energy from θ

```
V = m * g * L .* (1 .- cos.(_values))
```

Relative to the lowest point.

H3. Total energy

```
E = T .+ V
```

Should be nearly constant in the ideal, undamped model.

H4. Extracting data from solution

```
t        = sol.t
_values = sol[]
_values = sol[]
```

Symbolic indexing preserves the physical meaning of variables.

H5. Returning a `NamedTuple` for analysis

```
return (t=t, T=T, V=V, E=E)
```

Convenient for plotting and post-processing.

H6. Energy as diagnostic of solver quality

By inspecting $E(t)$, you can judge numerical drift and choose appropriate solvers or tolerances.

9 I. Plotting Helpers (`plot_energy`, `plot_phase`)

I1. Energy vs time plot

```
plt = plot(t, T, label="Kinetic Energy",
           xlabel="Time (s)", ylabel="Energy (J)",
           title="Pendulum Energy Over Time")
plot!(t, V, label="Potential Energy")
plot!(t, E, label="Total Energy")
```

Combines kinetic, potential, and total energy on one plot.

I2. `plot` vs `plot!`

```
plot(...)    # new figure
plot!(...)   # add to existing figure
```

Core idiom of `Plots.jl`.

I3. Phase-space (θ, ω) plot

```
_vals = sol[]
_vals = sol[]
plot(_vals, _vals,
     xlabel=" (rad)", ylabel=" (rad/s)",
     title="Pendulum Phase Space")
```

Visualizes trajectories in state space rather than time.

I4. Closed orbits for undamped pendulum

In the ideal case, phase portraits are closed loops (constant energy curves).

I5. Spiraling phase portrait with damping

Adding damping makes trajectories spiral toward the equilibrium in phase space.

I6. Encapsulating plotting in API

```
plot_energy(sol)
plot_phase(sol)
```

Small functions create a nicer user experience.

10 J. Damping Extension

J1. Add damping coefficient `c` as parameter

```
@parameters begin
    g = 9.81
    L = 1.0
    c = 0.1
end
```

J2. Damped pendulum equation

$$\dot{\omega} = -\frac{g}{L}\sin\theta - c\omega.$$

```
D() ~ -(g/L)*sin() - c*
```

J3. Expose damping in `simulate_pendulum`

```
function simulate_pendulum(; ..., c=0.1, ...)
    pend = Pendulum(g=g, L=L, c=c)
    ...
end
```

J4. Energy decay over time

With $c > 0$, $E(t)$ decreases; the phase portrait spirals inwards.

J5. Spiral phase portrait

`plot_phase(sol)` now shows spirals converging to $(\theta, \omega) = (0, 0)$.

11 K. Driven Pendulum Extension

K1. Introduce forcing amplitude and frequency

```
@parameters begin
    ...
    A = 0.0
     = 1.0
end
```

K2. Driven–damped equation

$$\dot{\omega} = -\frac{g}{L}\sin\theta - c\omega + A\sin(\Omega t).$$

```
D() ~ -(g/L)*sin() - c* + A*sin(*t)
```

K3. Expose `A` and in simulation API

```
function simulate_pendulum(; ..., A=0.0, =1.0, ...)
    pend = Pendulum(g=g, L=L, c=c, A=A, =)
    ...
end
```

K4. Long time spans for nonlinear effects

To see resonance or chaos, larger `tspan` is often needed, e.g. `(0.0, 300.0)`.

K5. Parameter regimes: periodic vs chaotic

By varying A and Ω, one can see transitions from periodic to quasi-periodic to chaotic behavior in the driven pendulum.

12 L. Poincaré Map Concepts and Utilities

L1. Driving period from Ω

$$T = \frac{2\pi}{\Omega}.$$

```
T = 2 /
```

L2. Sample times at multiples of T

```
t_samples = collect(t0:T:tend)
```

Used for stroboscopic (Poincaré) sampling.

L3. Skip transient periods

```
t_samples = t_samples[(nskip+1):end]
```

Focus on long-term behavior.

L4. Interpolate solution at sample times

```
u     = sol(tt)
_val = u[1]
_val = u[2]
```

Uses the solution's interpolation to evaluate off-grid values.

L5. Angle wrapping for clean plots

```
_vals = mod.(_vals .+ , 2) .-
```

Keeps angles in the interval $[-\pi, \pi]$.

L6. Poincaré point function

```
_vals, _vals = poincare_points(sol; =,
                                nskip=100, wrap=true)
```

Encapsulates sampling logic for reuse.

L7. Poincaré map plotting

```
scatter(_vals, _vals,
        xlabel=" (rad)", ylabel=" (rad/s)",
        title="Poincar Map",
        markersize=3, label="")
```

Visualizes the sampled points; structure reveals dynamics.

L8. Cloud vs discrete points

Discrete points (few distinct clusters) indicate periodic orbits; fuzzy clouds reflect chaotic behavior.

13 M. Learning and Reasoning Patterns Embedded in the Plan

M1. Rhetorical questions as checkpoints

Transform the tutor's questions into self-check prompts (e.g., "Why do we use $\sin(\theta)$? Why first-order form?") at each stage.

M2. From intuition → math → code

Pipeline:

(a) Physical reasoning.

(b) Mathematical equations.

(c) MTK symbolic code.

(d) Package-level APIs.

M3. Definitions vs laws vs constraints

Classify each equation:

- Kinematic definitions,
- Dynamic laws,
- Constraints linking everything.

M4. Thinking in system structure graphs

Variables and equations form a bipartite graph. MTK uses this for structural analysis and index reduction.

M5. Separating modeling from package engineering

First build the physical model; then wrap it in `@mtkmodel`; then build the package, tests, and docs.

M6. Use of defaults for fast exploration

Defaults (e.g. $g = 9.81, L = 1.0$) allow `Pendulum()` and `simulate_pendulum()` to work immediately.

M7. Layered abstraction: low-level MTK → high-level helpers

Example:

- `ODESystem` → `simulate_pendulum`
- `sol` → `compute_energy` → `plot_energy`

M8. Separation of concerns in files

`pendulum.jl` (model), `simulate.jl` (running), `energy.jl` (analysis), `plots.jl` (visualization), `poincare.jl` (chaos tools).

M9. Naming aligned with physics

Use symbolic names close to the equations:

```
, , g, L, c, A,
```

M10. Building from minimal model to advanced features

Start with the simplest undamped model; then add damping, forcing, energy tools, and finally chaos-related analysis.

M11. Simulation as experiment, not just plotting

Change one parameter at a time and observe energy, phase space, and Poincaré maps to develop physical intuition.

M12. Design APIs for the way you want to think

```
sol = simulate_pendulum(A=1.2, =0.66, tspan=(0,500))
plot_poincare(sol; =0.66)
```

Design the interface around how you naturally want to explore the system.

M13. Using the pendulum journey as template

The same approach (connectors, first-principles, MTK, package, analysis) can be applied to other domains like hydropower (pressure–flow), turbines, and multi-physics systems.